ELSEVIER

Theoretical Computer Science 231 (2000) 89–101

Theoretical Computer Science

www.elsevier.com/locate/tcs

language membership ☆

J.-L. Ponty

Laboratoire d'Informatique Fondamentale et Appliquée de Rouen, Faculté des Sciences et des Techniques, 76821 Mont-Saint-Aignan Cedex, France

Abstract

We present a new algorithm to determine whether a given word belongs to the language denoted by a regular expression. It is based on our ZPC representation of the Glushkov automaton of a regular expression. This procedure requires a specific representation of the Glushkov automaton of the expression. The representation is computed in linear time and space using the ZPC algorithm designed by Ziadi et al. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Automata theory; Glushkov construction; Regular expressions; String matching

1. Introduction

Regular expressions and finite automata play an important role in the design of many applications such as lexical analyzers, text editors, word processors, electronic dictionary, filters and UNIX utilities (for example sed, grep and awk). The feature these fundamental tools share is that they need to check whether or not a word belongs to a given language. More precisely, this problem can be formulated as follows: given an alphabet Σ, a word w and a regular expression E on Σ, we have to decide whether or not the word w belongs to the language $L(E)$. It is classically solved by constructing an ε-automaton recognizing $L(E)$ and verifying whether there exists a successful path labelled by w in this automaton [25, 3, 5]. The first step is in $O(|E|)$ time and space, whereas the second step is in $O(|E| \times |w|)$ time and in $O(|E|)$ space, where $|E|$ is the size of E.

This paper presents a new algorithm based on an original representation of the Glushkov automaton of the expression E. The first step of our approach is the

☆ This work is a contribution to the Automate software development project carried on by A.I.A. Working Group (Algorithmics and Implementation of Automata), L.I.F.A.R. laboratory (contact: {Jean-Marc.Champarnaud, Djelloul.Ziadi}@dir.univ-rouen.fr).

E-mail address: Jean-Luc.Ponty@dir.univ-rouen.fr (J.-L. Ponty)

PII: S0304-3975(99)00019-5

recognizer			construction of the recognizer		membership test	
type	*states*	*transitions*	*time*	*space*	*time*	*space*
DFA	$N \leqslant 2^n$	$\|\delta\| \leqslant \|\Sigma\|N$	$O(2^n)$	$O(2^n)$	$O(\|w\|)$	$O(1)$
NFA	n	$\|\delta\| \leqslant \|\Sigma\|n^2$	$\Omega(n^{p\geqslant 2})$	$\Omega(n^2)$	$O(n^2 \times \|w\|)$	$O(n)$
Thompson	$n \leqslant 2\|E\|$	$\|\delta\| \leqslant 4\|E\|$	$O(n)$	$O(n)$	$O(n \times \|w\|)$	$O(n)$
Glushkov ZPC struct.	$n = \lceil \|E\|/2 \rceil + 1$	Follow links $\leqslant \|E\|$	$O(n)$	$O(n)$	$O(n \times \|w\|)$	$O(n)$

Fig. 1. Different recognizers and their performances.

construction of the ZPC representation [23, 27] of the Glushkov automaton of the expression E. This representation can be computed in $O(|E|)$ time and space. The second step (the PZC function) consists in simulating the search of a successful path labelled by w, on this representation. Our claim is that this second step is in $O(|E| \times |w|)$ time and $O(|E|)$ space. So the whole procedure has the same time and space complexity as algorithms using ε-transitions. It is an interesting result for automata implementation, since asynchronous automata are usually superfluously larger than ε-free automata [18]. Furthermore, it is interesting for theoretical computer science to see that such a data structure as the ZPC representation leads to efficient algorithms to solve classical problems as language membership or determinization.

Let us terminate this introduction by noticing that the problem we are concerned with and pattern matching [10] are very close topics. Given an alphabet Σ, a word (*the text*) on Σ, and a set of words (*the pattern*) on Σ, pattern matching consists in locating an occurrence (or occurrences) of an arbitrary word belonging to *the pattern* in *the text*. If the pattern is denoted by a regular expression, a classical solution is to check whether or not prefixes of the text belong to the language $\Sigma^*(L(E) \setminus \{\varepsilon\})$.

The next section is a comparative study of several well-known membership test procedures. Section 3 recalls some definitions and notations used in the description of the ZPC and PZC algorithms. Section 4 outlines the main features of the ZPC algorithm. Section 5 presents the PZC algorithm and proves its complexity. Further developments are mentioned in the conclusion.

2. An overview of membership test procedures

The standard method first computes an automaton $\mathscr{A}$ recognizing $L(E)$ and then exhibits a successful path labelled by w in $\mathscr{A}$, if any. According to the type of the recognizer (such as is-deterministic), its construction and the membership test are more or less efficient. The following array summarizes the performances associated to four types of recognizers:

- DFA: a deterministic automaton [1].

- NFA : a nondeterministic and ε-free automaton.
- Thompson: the standard ε-automaton of the regular expression E [25, 5],
- Glushkov: the Glushkov automaton of the regular expression E, a nondeterministic and ε-free automaton [12, 6, 7, 9, 23].

The **size** $|E|$ of a regular expression is defined to be the total number of appearances of operators and symbols in E, whereas the number of appearances of symbols of Σ in E is denoted by $\|E\|$. The set of transitions of an automaton is denoted by δ. The DFA recognizer is supposed to result from the determinization of an NFA with n states.

Let us recall that the number of states of the DFA recognizer is exponential on the size of the expression, in the worth case. So DFA recognizers are generally used in specific cases, as in matching one word or a finite set of words in a text. In these two situations linear or sublinear (on the sum of the lengths of the pattern and of the text) tests have been designed by Morris and Pratt [1], Knuth et al. [17] and Simon [24, 13] for a one-word pattern, and by Aho and Corašick [2] for a finite set of words. The last-mentioned algorithm has been extended by Mohri [20] in the case of arbitrary automata. Let us mention that some of the tools of the UNIX operating system such `egrep` are based on such DFA recognizers [1], although the DFA may not be completely computed. Finally, let us point out that Brüggemann-Klein and Wood [8] have characterized languages which can be denoted by a regular expression whose Glushkov automaton is a DFA.

3. Definitions and notations

We shall limit ourselves to definitions involved by the description of the next algorithms. For further details about regular languages and finite automata, Refs. [14, 26, 11, 21] should be consulted.

A finite automaton is a 5-tuple $\mathscr{M} = (Q, \Sigma, I, F, \delta)$ where Q is a (finite) set of states, Σ is a (finite) alphabet, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of terminal states, and δ is the transition function. A deterministic finite automaton (DFA) has a unique initial state and arrives in a unique state (if any) after scanning a symbol of Σ. Otherwise the automaton is nondeterministic (NFA). The language $L(\mathscr{M})$ recognized by the automaton $\mathscr{M}$ is the set of words of Σ^* whose scanning makes $\mathscr{M}$ arrive to a terminal state.

A regular expression over an alphabet Σ is generated by recursively applying operators '$+$' (union), '$\cdot$' (concatenation product) and '$*$' (Kleene star) to atomic expressions (every symbol of Σ, the empty word and the empty set). A language is regular if and only if it can be denoted by a regular expression.

The Kleene theorem [16] states that a language is regular if and only if it is recognized by a finite automaton. Computing the Glushkov automaton of a regular expression [12] is a constructive proof of this theorem.

Glushkov algorithm works on a linearized expression E' deduced from E by ranking every symbol occurrence with its position in E.

For example:

$$\text{if } E=(a+(a+b)^*a)(a+b)^* \quad \text{then } E'=(a_1+(a_2+b_3)^*a_4)(a_5+b_6)^*.$$

The set of positions of E is denoted by $Pos(E)$. The number of positions, called the alphabetic width of E [4] is denoted by $\|E\|$. The application $\chi : Pos(E) \to \Sigma$ maps every position to its value in Σ.

Algorithm Glushkov(E)

1. Linearize the expression E. Result is expression E'.
2. Compute the following sets:
   - $Null_E$ which is $\{\varepsilon\}$ if $\varepsilon \in L(E)$ and $\emptyset$ otherwise.
   - $First(E)$, the set of positions that match the first symbol of some word in $L(E')$.
   - $Last(E)$, the set of positions that match the last symbol of some word in $L(E')$.
   - $Follow(E, \forall x \in Pos(E)$: the set of positions that follow the position x in some word of $L(E')$.
3. Compute the Glushkov automaton of E, $\mathcal{M}_E=(Q,\Sigma,s_I,F,\delta)$ where:
   - $Q=Pos(E)\cup\{s_I\}$
   - $\forall a \in \Sigma, \delta(s_I,a)=\{x \in First(E) \mid \chi(x)=a\}$
   - $\forall x \in Q, \forall a \in \Sigma, \delta(x,a)=\{y \mid y \in Follow(E,x) \text{ and } \chi(y)=a\}$
   - $F=Last(E)\cup Null_E\cdot\{s_I\}$

A straightforward implementation of this algorithm, based on a recursive computation of the sets $Null_E, First(E), Last(E)$ and $Follow(E,x)$ leads to an $O(\|E\|^3)$ time complexity. Brüggemann-Klein [7], Chang and Paige [9], and Ziadi et al. [23, 27] have designed quadratic variants. We describe the latter briefly in Section 4.

4. The ZPC algorithm

The ZPC algorithm first converts a regular expression E into a couple of forests deduced from its syntax tree $T(E)$. These forests respectively encode the Last sets and the First sets related to the subexpressions of E. The transition function of the Glushkov automaton of E appears as a collection of links going from the Last forest to the First forest.

In the following we shall write $First(v)$ instead of $First(E_v)$, where E_v is the subexpression of E related to the node v of $T(E)$. This convention holds for $Last(v)$, $Follow(v,x)$ and $Null_v$.

Given a node v of $T(E)$, we shall denote by λ the node corresponding to v in the Last forest, $TL(E)$, and by φ the node corresponding to v in the First forest, $TF(E)$.

Let us sketch out the ZPC algorithm.

Algorithm ZPC(E)

1. Compute the syntax tree $T(E)$.
2. Compute the forests $TL(E)$ and $TF(E)$.
3. Compute the set of follow links going from $TL(E)$ to $TF(E)$.
4. Remove redundant follow links.
5. Compute the transition table of $\mathscr{M}_E$.

Steps 1 – 4 compute the so-called ZPC(E) representation of E. We shall only describe these steps, on which is based our membership test.

The Last forest $TL(E)$ is a copy of $T(E)$, where a link going from a node labelled '$\cdot$' to its left child is deleted if its right child does not recognize ε. Thus the property $Last(F)\cdot(G) = Last(G) \cup Null_G \cdot Last(F)$ is satisfied. Furthermore, each node of $TL(E)$ points to its leftmost and rightmost leaves, and leaves in the same Last set are linked.

The First forest $TF(E)$ is computed in a similar way, by deleting a link going from a node labelled '$\cdot$' to its right child, if its left child does not recognize ε, w.r.t. the property: $First((F)\cdot(G)) = First(F) \cup Null_F \cdot First(G)$.

The two forests are connected as follows. If a node of $TL(E)$ is labelled by '$\cdot$', its left child is linked to the right child of the corresponding node in $TF(E)$. If a node is labelled by '$*$', its child is linked to the child of the corresponding node in $TF(E)$. Such links are called *follow links*.

Notice that a follow link encodes the cartesian product of a Last set by a First set, and that the transition function δ is the union of such cartesian products. Two products are either disjoint, or included in each other. Redundant products are eliminated by a recursive procedure. Finally, the representation ZPC(E) is such that a transition is encoded in a unique follow link.

Example 4.1. Consider the expression $E = (a + (a + b)^* a)(a + b)^*$. The linearized expression is $E' = (a_1 + (a_2 + b_3)^* a_4)(a_5 + b_6)^*$. So we can build the representation ZPC as shown in Fig. 2.

It is convenient to process the expression $E' = \$((a_1 + (a_2 + b_3)^* a_4)(a_5 + b_6)^* \#)$, where \$ and # are two distinguished positions. The position \$ is associated to the initial state of $\mathscr{M}_E$ and is involved in the follow link: $\{\$\} \times First(E)$. The position # is reached from positions which belong to $Last(E)$ by scanning the end of the input word; it appears only in the follow link $Last(E) \times \{\#\}$. Notice that \$ is involved in this link too if $\$ \in Last(E)$, i.e. if $L(E)$ recognizes ε.

5. The PZC membership test

The PZC algorithm parses the word w on the two forests $TL(E)$ and $TF(E)$ according to properties of the follow links. Let us recall that δ is the transition function of the

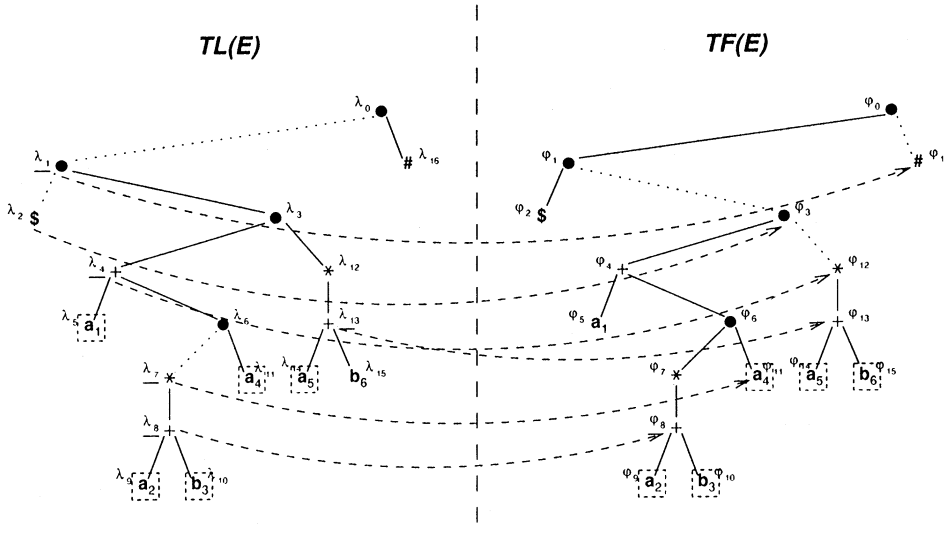

Fig. 2. ZPC(E) for $E' = (a_1 + (a_2 + b_3)^* a_4)(a_5 + b_6)^*$.

Glushkov automaton of the regular expression E. For every set X of positions of E and for every symbol s of Σ, we shall denote by $\delta(X,s)$ the set of positions which are reached from any position of X by reading the input symbol s. We have

$$\delta(X,s) = \left(\bigcup_{x \in X} Follow(E,x) \right) \cap Pos_s(E),$$

where $Pos_s(E)$ is the set of positions y of E such that $\chi(y) = s$.

The PZC algorithm can be stated as follows:

Let w be the input word, with $w = w_1 \dots w_p$ and $w_i \in \Sigma$, $\forall i \in [1, p]$. Let $X_0 = \{\$\}$ and let us compute the list $(X_0, X_1, \dots, X_q)$ of sets of positions such that $X_i = \delta(X_{i-1}, w_i)$ if $X_{i-1} \neq \emptyset$, $\forall i = 1$ to $q \leqslant p$.

Finally we have:

$$w \in L(E) \Leftrightarrow \begin{cases} q = p \\ X_p \cap Last(E) \neq \emptyset. \end{cases}$$

This algorithm can be implemented in $O(||E|| \times |w|)$ time and $O(||E||)$ space, according to the following lemmas.

Lemma 5.1. *The size of the syntax tree $T(E)$ is bounded by $3||E|| - 2$.*

This follows from the assumption that E does not contain any subexpression such as $((F^*)^*)$ and that ε occurs only in union subexpressions. We have proved [22] by induction on the size of E that the number of non-redundant follow links is bounded by $2||E|| - 1$.

```
function in-lasts(TL(E), X, m): set of nodes of TF(E)
    /* TL(E) is the forest of Lasts of E */
    /* X is a set of positions */
    /* m is a mark */
    begin
        Λ ← ∅
        foreach x in X do
            /* let λ' be the leaf of TL(E) labelled by x */
            λ ← λ'
            repeat
                mark λ by m
                if λ is the head of a follow link
                then Λ ← Λ ⊎ {λ}
                fi
                λ ← ancestor(λ)
            until λ is marked by m
        od
        return Λ
    end
```

Fig. 3. The algorithm in-lasts.

Lemma 5.2. *Let X be a set of positions of E. Let Λ be the set of nodes λ of $TL(E)$ such that*
1. *The set $Last(\lambda)$ contains at least one element of X.*
2. *There exists a follow link with node λ as head.*

Our claim is that the set Λ can be computed in $O(||E||)$ time and space.

The *in-lasts* function (Fig. 3) inspects the path going from any position x of X to the root of the tree of $TL(E)$ x belongs to. Let λ be the current node. If λ is marked, the inspection concerning position x is stopped. Otherwise, λ is marked, and it is added to the set Λ if it is the head of a follow link.

Thanks to the marking of nodes, each head of follow link is added only once to Λ. Moreover, each node λ is visited at most d times where d is the number of children of λ in $TL(E)$; thus, by Lemma 5.1, *in-lasts* function is in $O(||E||)$ time.

Assuming that the ancestor of the root of a tree in $TL(E)$ is this root itself, we can write the *in-lasts* function as follows.

Example 5.1. Let $E = \$((a_1 + (a_2 + b_3)^* a_4)(a_5 + b_6)^*)\#$, and consider the set of positions $X = \{1, 2, 3, 4, 5\}$. We run in-lasts function on the representation $ZPC(E)$ of

Fig. 2. Follow links starting from λ_4 and λ_1 (resp. λ_8 and λ_7) are marked when inspecting the path going from the position 1 (resp. 2). The path going from the position 3 (resp. 4) is scanned until marked node λ_8 (resp. λ_4) is reached. Concerning the position 5, λ_{13} is added to Λ and the inspection is stopped by λ_3 which is marked. So we get $\Lambda = \{\lambda_4, \lambda_1, \lambda_8, \lambda_7, \lambda_{13}\}$.

Lemma 5.3. *Consider a set of follow links, and let Φ be the set of nodes φ of $TF(E)$ which are tails of these links. Our claim is that the set*

$$Y = \bigcup_{\varphi \in \Phi} First(\varphi)$$

can be computed in $O(||E||)$ *time and space.*

The *in-firsts* function (Fig. 4) initializes a set Φ' to the empty set and performs a preorder tree walk of each tree of the forest $TF(E)$. Let φ be the current node. If $\varphi \in \Phi$ then the traversal of the subtree rooted at φ is not processed and φ is added to the set Φ'. Finally, the set Φ' is such that $Y = \biguplus_{\varphi \in \Phi'} First(\varphi)$, hence the result.

Example 5.2. Let $E = \$((a_1 + (a_2 + b_3)^* a_4)(a_5 + b_6)^*)\#$, and consider the set of nodes of the forest $TF(E)$, $\Phi = \{\varphi_8, \varphi_{11}, \varphi_{12}, \varphi_{13}\}$. We run the *in-firsts* function on the representation $ZPC(E)$ of Fig. 2. Notice that this example is independent of the previous one. Inspecting the tree rooted at φ_3 yields φ_8 and φ_{11}. In the tree rooted at φ_{12} only the root is inspected, adding φ_{12} to Φ'. Finally, we get $\Phi' = \{\varphi_8, \varphi_{11}, \varphi_{12}\}$ and $Y = \{2,3\} \uplus \{4\} \uplus \{5,6\}$.

Remark. (1) The test $\varphi \in \Phi$ is realized via a marking in the forest $TF(E)$ of the nodes belonging to Φ.

(2) The function *in-firsts* can be improved by coding the index of the tree in the name of nodes φ of $TF(E)$ and handling an array of pointers on the roots of the trees. Thus a tree which does not contain any element of Φ is not inspected, neither is a tree which contains exactly one element φ of Φ. In the latter case, just add φ to the set Φ'.

Finally, *in-lasts* and *in-firsts* functions are used to compute in $O(||E||)$ time and space each of the sets $X_i = \delta(X_{i-1}, w_i)$, for $i \in [1, q]$. $O(||E||)$ space is common to each of the steps. So we can state the following theorem:

Theorem 5.1. *Let Σ be an alphabet. Let E be a regular expression on Σ, and $||E||$ be its alphabetic width. Let w be a word of Σ^*. Let $ZPC(E)$ be the representation of the Glushkov automaton of E which is computed by the ZPC algorithm. The membership*

```
function in-firsts(TF(E), Φ): set of positions
    /*TF(E) is the First forest of E */
    /* Φ is a set of nodes of TF(E) */
    procedure in-tree(φ, Φ, Φ′)
        /* φ is a node of TF(E) */
        /* Φ and Φ′ are sets of nodes of TF(E) */
        begin
            if φ ≠ nil
            then  if φ ∉ Φ
                  then  switch (symbol(φ))
                        begin
                            case '∗':
                                in-tree(φ_c, Φ, Φ′)
                            case '·','+',x:
                                in-tree(φ_l, Φ, Φ′)
                                in-tree(φ_r, Φ, Φ′)
                        end
                  else Φ′ = Φ′ ⊎ {φ}
                  fi
            fi
        end
    begin /* function in-firsts */
        Φ′ ← ∅
        foreach T in TF(E) do
            /* let φ_T be the root of the tree T */
            in-tree(φ_T, Φ, Φ′)
        od
        Y ← ∅
        foreach φ in Φ′ do
            Y ← Y ⊎ First(φ)
        od
    return Y
    end
```

Fig. 4. The algorithm in-firsts.

of w to the language $L(E)$ can be decided in $O(||E|| \times |w|)$ *time and* $O(||E||)$ *space through ZPC(E) representation.*

Example 5.3. Let $E = \$((a_1+(a_2+b_3)^*a_4)(a_5+b_6)^*)\#$, and consider the string $w = ab$. We verify from Figs. 6 and 7 that w belongs to $L(E)$.

```
function PZC(E, w): boolean
    /* E is a regular expression */
    /* w is a word */
    begin
        /* compute the ZPC(E) representation of E */
        /* process w through the forests TL(E) and TF(E) */
        X ← {$}
        i ← 0
        while ((i < |w|) and (X ≠ ∅)) do
            i ← i + 1
            Λ ← in-lasts(TL(E), X, i)
            Φ ← ∅
            foreach λ in Λ do
                /* consider the follow link(λ, φ) */
                Φ ← Φ ∪ {φ}
            od
            X ← ∅
            Y ← in-firsts(TF(E), Φ)
            foreach x in Y do
                if χ(x) = w[i] then X ← X ∪ {x}
            od
        od
        return ((i = |w|) and (X ∩ Last(E) ≠ ∅))
    end
```

Fig. 5. The PZC algorithm.

i	w_i	Λ	Φ	Φ'	Y	X
0		$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{\$\}$
1	a	$\{\lambda_2\}$	$\{\varphi_3\}$	$\{\varphi_3\}$	$\{1,2,3,4\}$	$\{1,2,4\}$
2	b	$\{\lambda_4,\lambda_1,\lambda_8,\lambda_7\}$	$\{\varphi_{12},\varphi_{16},\varphi_8,\varphi_{11}\}$	$\{\varphi_{12},\varphi_{16},\varphi_8,\varphi_{11}\}$	$\{5,6,\#,2,3,4\}$	$\{6,3\}$

Fig. 6. Running PZC on $\$((a_1+(a_2+b_3)^*a_4)(a_5+b_6)^*)\#,ab$.

6. Conclusion

We have shown that ε-transitions are avoidable when performing regular language membership test. To ensure that the PZC algorithm runs in time and space linear in the size of the expression, it uses a natural partition of the transitions of the Glushkov automaton of the expression, rather than the automaton itself. Notice that the alternative which consists in computing an ε-free NFA recognizer that is *as small as possible*

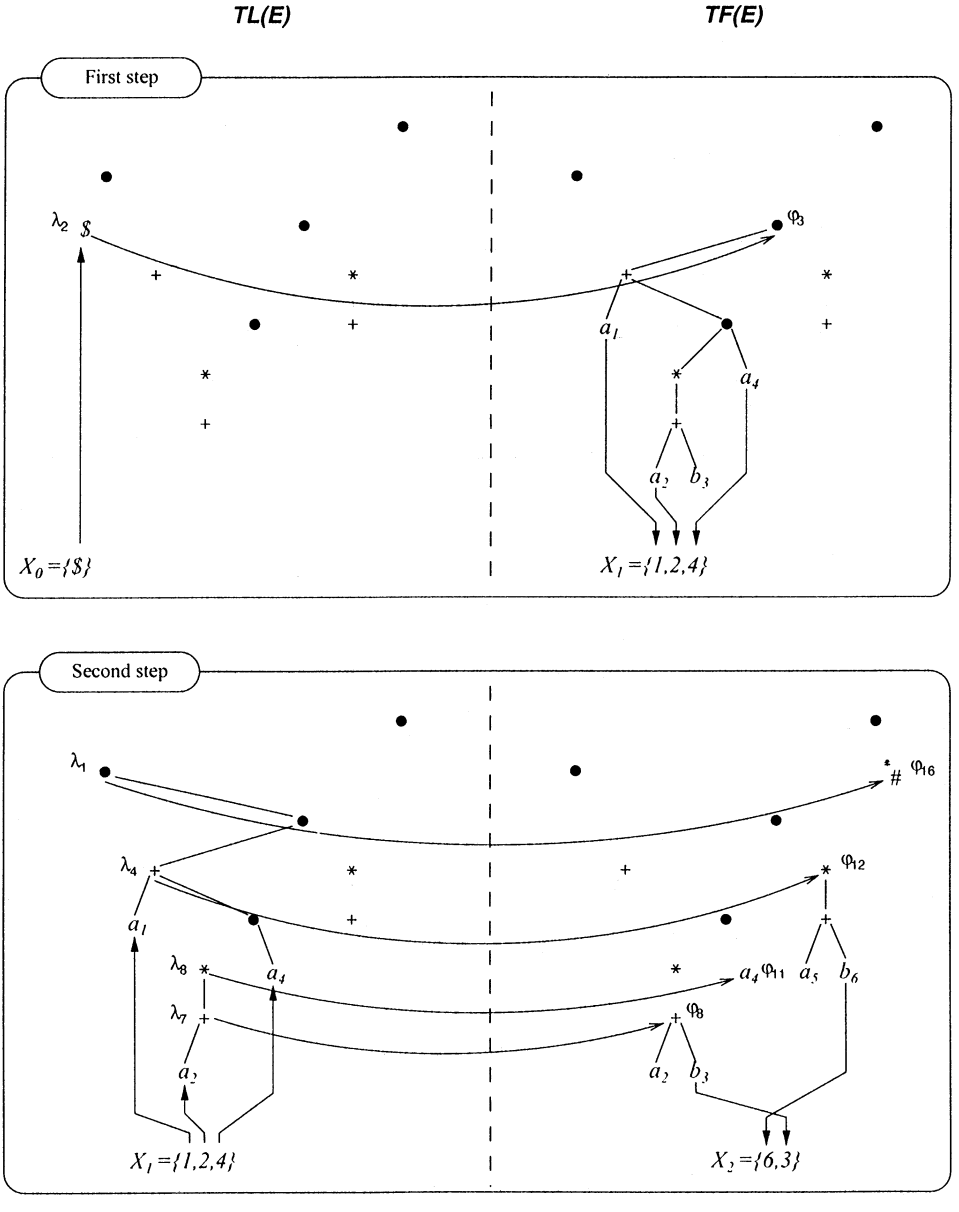

Fig. 7. Using the *ZPC* structure to compute PZC(E, w).

with respect to the number of states or the number of transitions, following Mirkin [19] or Antimirov [4], is hopeless. Indeed, Hromkovič et al. have pointed out [15] that there exist regular expressions such that any ε-free automaton recognizing them has at least $O(|E|\log(|E|))$ transitions, which yields a membership test that runs in at least $O(|E|\log(|E|) \times |w|)$ time. We are currently investigating properties of the ZPC

representation of Glushkov automata to improve the determinization and minimization algorithms of such NFAs.

Acknowledgement

I would like to thank the referees for their suggestions and remarks in order to improve this paper. Moreover I am most grateful to Derick Wood for his friendly help.

References

[1] A.V. Aho, Algorithms for finding patterns in strings, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Algorithms and Complexity, vol. A, Ch. 5, Elsevier, Amsterdam, 1990, pp. 255–300.
[2] A.V. Aho, M.J. Corašick, Efficient string matching: an aid to bibliographic search, Commun. ACM 18(6) (1975) 333–340.
[3] A.V. Aho, R. Sethi, J.D. Ullman, Compilers – Principles, Techniques and Tools, Addison-Wesley, Reading, MA, 1986.
[4] V. Antimirov, Partial derivatives of regular expressions and finite automaton constructions, Theoret. Comput. Sci. 155 (1996) 291–319.
[5] D. Beauquier, J. Berstel, P. Chrétienne, Éléments d'Algorithmique, Masson, Paris, 1992.
[6] G. Berry, R. Sethi, From regular expressions to deterministic automata, Theoret. Comput. Sci. 48(1) (1986) 117–126.
[7] A. Brüggemann-Klein, Regular expressions into finite automata, Theoret. Comput. Sci. 120(1) (1993) 197–213.
[8] A. Brüggemann-Klein, D. Wood, Deterministic regular languages, in A. Finkel, M. Jantzen (Eds.), Proc. 9th Annual Symp. on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, vol. 577, Springer, Berlin, 1992, pp. 173–184.
[9] C.-H. Chang, R. Paige, From regular expressions to DFA's using NFA's, in A. Apostolico, M. Crochemore, Z. Galil, U. Manber (Eds.), Proc. 3rd Annual Symp. on Combinatorial Pattern Matching, Lecture Notes in Computer Science, vol. 664, Tucson, AZ, Springer, Berlin, 1992, pp. 90–110.
[10] M. Crochemore, C. Hancart, Automata for matching patterns, in: G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, vol. II, Linear Modeling: Background and Application, Springer, Berlin, 1997, pp. 399–462
[11] M.D. Davis, R. Sigal, E.J. Weyuker, Computability, Complexity, and Languages, Fundamentals of Theoretical Computer Science, Academic Press, New york, 1994.
[12] V.M. Glushkov, The abstract theory of automata, Russian Math. Surveys 16 (1961) 1–53.
[13] C. Hancart, On Simon's string searching algorithm, Inform. Process. Lett. 47(2) (1993) 95–99.
[14] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages and Computation, Addison-Wesley, Reading, MA, 1979.
[15] J. Hromkovič, S. Seibert, T. Wilke, Translating regular expressions into small ε-free nondeterministic finite automata, in: R. Reischuk, M. Morvan (Eds.), STACS 97, 14th Annual Symp. on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, vol. 1200, Lübeck, Germany, February 27-March 1, Springer, Berlin, 1997, pp. 55–66.
[16] S. Kleene, Representation of events in nerve nets and finite automata, in: Automata Studies, Ann. Math. Stud., vol. 34, Princeton Univ. Press, Princeton, 1956, pp. 3–41.
[17] D.E. Knuth, J.H. Morris, V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6(2) (1977) 323–350.
[18] O. Matz, A. Miller, A. Potthoff, W. Thomas, E. Valkena, Report on the program AMoRE, Report, Institut für informatik und praktische mathematik, Christian-Albrechts Universität, Kiel, 1995.

[19] B.G. Mirkin, An algorithm for constructing a base in a language of regular expressions, Eng. Cybernet. 5 (1966) 110–116.
[20] M. Mohri, Matching patterns of an automaton, in: Z. Galil, E. Ukkonen (Eds.), Proc. 6th Annual Symp. on Combinatorial Pattern Matching, Lecture Notes in Computer Science, vol. 937, Espoo, Finland, Springer, Berlin, 1995, pp. 286–297.
[21] D. Perrin, Finite automata, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Formal Models and Semantics, vol. B, Elsevier, Amsterdam, 1990, pp. 1–57.
[22] J.-L. Ponty, Algorithmique et Implémentation des Automates, Contribution au Développement du Logiciel AUTOMATE, Ph.D. Thesis, LIR, Université de Rouen, France, 1997, Rapport LIR TH97.03.
[23] J.-L. Ponty, D. Ziadi, J.-M. Champarnaud, A new quadratic algorithm to convert a regular expression into an automaton, in: D. Raymond, D. Wood, S. Yu (Eds.), Automata Implementation: First International Workshop on Implementing Automata, WIA'96, Lecture Notes in Computer Science, vol. 1260, London, Ontario, Springer, Berlin, 1997, pp. 109–119.
[24] I. Simon, Piecewise testable events, in Proc. 2nd GI Conf., Lecture Notes in Computer Science, vol. 33, Springer, Berlin, 1975, pp. 214–222.
[25] K. Thompson, Regular expression search algorithm, Commun. ACM 11(6) (1968) 419–422.
[26] D. Wood, Theory of Computation, Wiley, New York, 1987.
[27] D. Ziadi, J.-L. Ponty, J.-M. Champarnaud, Passage d'une expression rationnelle à un automate fini non-déterministe, Bull. Belg. Math. Soc. 4 (1997) 177–203.